

FORTH, MUMPS, etc...

JUIN 1986

20 Fr

EST-CE QUE
ÇA NOURRIT
SON HOMME ?

JIM DAVIS

EDITORIAL

Au cours d'une récente discution avec un de nos rédacteurs d'articles, nous avons été amenés à débattre des avantages et inconvénients des divers langages utilisés sur les microordinateurs. Bien entendu, chacun défendait son point de vue sur la partie le concernant, par passion d'abord, par conviction ensuite.
Mais tout le monde semblait obsédé par les performances à la compilation ou à l'exécution des programmes en fonction de tel ou tel langage. Il n'y eut pas de conclusion, chacun restant convaincu du bien fondé de sa démarche personnelle.
En fait, le langage le plus pratique, c'est celui que vous maitrisez. Peu importe les performances, à condition qu'il remplisse les fonctions auquelles vous le destinez, même s'il s'agit de BASIC. Mr SCHERER m'a dit, à juste titre, que dans certains cas une brouette est plus utile qu'une voiture de course, surtout quand il s'agit de transporter vingt kilos de pommes de terre sur vingt mètres.
Si l'on applique les principes d'une pensée aussi profonde à l'univers de l'informatique, on arrive à la conclusion que tout est bon à prendre, qu'il s'agisse des systèmes comme des langages et des programmes. Il suffit que ceuxci servent. Effectivement, combien d'utilisateurs exploitent toutes les possibilités offertes par leur système. Il en est de même des langages.
Vautil mieux programmer avec des langages à tout faire, véritables dinosaures fossilisés par leur rigidité, ou exploiter des modules extrêmement compacts, semblables à des boîtes à outils pour passionnés du bricolage ?
Le débat reste ouvert entre partisans du meccano et tenants du produit fini: Forth ou dBASE II, APL ou ASSEMBLEUR, etc.. ?
Alors faisons plaisir à chacun. Voici une nouvelle brouettée de trucs pour vous faire passer des vacances studieuses à l'ombre de vos disquettes. Ou alors prendrezvous place à bord de notre MUMPS six cylindres et irez-vous par les chemins menants au confort des gros systèmes ? Quel que soit votre destination, revenezen avec une petite provision de soleil et mitonnez nous quelques petits sujets juteux à faire partager à nos adhérents impatients d'apprendre.

SOMMAIRE

ASSOCIATION JEDI 8, rue Poirier de Narçay 75014 PARIS
Tel: (1) 45.42.88.90 (de 10h à 18h)

FORTH Les en-têtes séparés

par Klaus SCHLEISIEK

La mémoire des systèmes en RAM peut être utilisée plus efficacement au moyen d'une "zone de dictionnaire des symboles", permettant de se débarrasser après compilation des mots et/ou des champs nom et lien qui ne sont nécessaires qu'au moment de la compilation. L'utilisation croissante de ces techniques donnera un meilleur emploi de la mémoire et encouragera la création de définitions plus courtes et plus nombreuses puisque la pénalisation en espace mémoire dûe aux champs nom et lien n'existe plus.

Au cours d'un projet de deux ans, j'ai développé quelques outils permettant une compression significative du code dans les systèmes en RAM. Les méthodes utilisées sont exposées ci-dessous, dans un ordre quelque peu historique.

Le but était le développement d'un système sonore contrôlé par crayon optique, devant permettre la commande d'un certain nombre de synthétiseurs par pointage du crayon sur des points, des potentiomètres lumineux, etc,... d'un écran vidéo. Aucune intervention au clavier n'était prévue. Un "point" était assemblé par compilation d'un mot associant les informations suivantes:

A) - la forme du point comme adresse d'un caractère programmable;

B) - l'emplacement du point sur l'écran comme adresse par rapport au coin supérieur gauche de l'écran;

C) - en option, une chaîne soit de texte, soit de caractères graphiques pouvant être affichée au-dessus, au-dessous, ou d'un côté ou l'autre du point.

Ainsi, chaque "point" a un double but. D'une part, il décrit une partie de l'affichage devant clignoter sur l'écran. D'autre part, il fournit la clé d'une grande déclaration CASE qui associe le point et la fonction à accomplir lorsque le crayon y est pointé. En d'autres termes, les définitions des points eux-mêmes ne sont utilisées qu'à la compilation.

Les définitions des points furent utilisées pour créer une "image" dense de la définition faisant clignoter le dessin à l'écran, cependant que les adresses des emplacements des points servaient comme clés dans CASE. Pour gagner en efficacité, je désirais placer un "mécanisme" permettant la présence de "symboles" au moment de la compilation, et qui pourrait ensuite les "jeter" pour libérer la mémoire. Par "symbole" je désigne toute définition permise par Forth nécessaire seulement à la compilation. D'où l'idée de diviser le dictionnaire en "dictionnaire principal" et "dictionnaire des symboles".

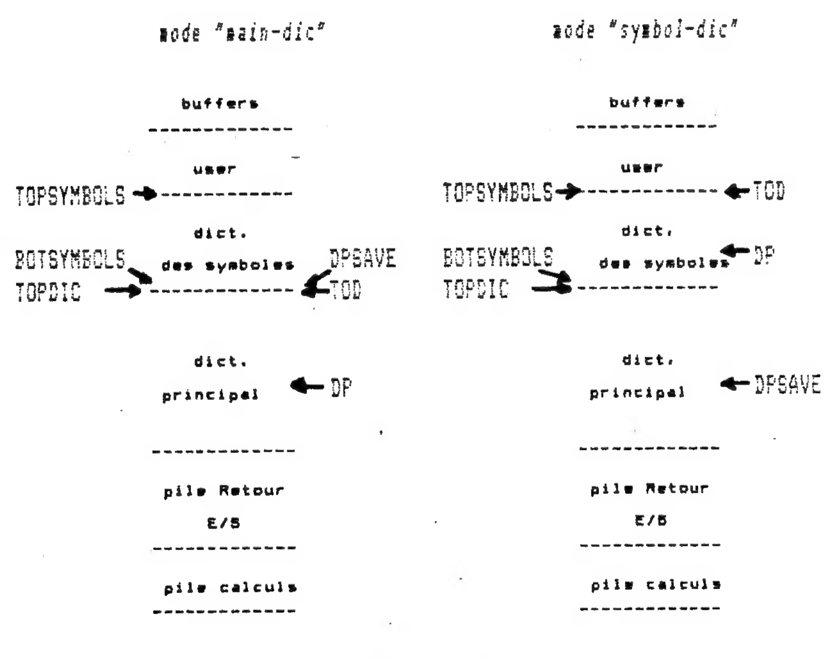


Figure 1

La figure 1 montre l'organisation de la mémoire issue de ce schéma dans le cas particulier d'un 6502.

Je m'aperçus vite que la plupart des mots définis dans mes programmes ne seraient plus jamais utilisés après compilation, et commençais à réfléchir à la possibilité de placer le nom et le champ de lien (l'en-tête) dans un dictionnaire des symboles, et de ne compiler dans le dictionnaire principal que les champs code et paramètres.

Celà signifiait aller et venir entre un état compilant les en-têtes dans le dictionnaire des symboles et un autre état les compilant de la manière habituelle. Le passage d'un état à l'autre est fait par la variable HEADFLG (écran #23), qui est activée par DROP-HEADS, désactivée par COMPILE-HEADS (écran #23). L'état de HEADFLG modifie à son tour le comportement de CREATE (écran #24).

Une complication apparait alors, car l'utilisation de HEADFLG interfère avec le mécanisme du dictionnaire des symboles: si vous êtes en train de compiler dans le dictionnaire principal, vous voulez que les en-têtes soient compilées dans le dictionnaire des symboles, mais si vous compilez dans celui-ci, vous voulez que les en-têtes y aillent également.

En d'autres mots, dans le premier cas le corps d'une définition sera séparé de la "tête", alors que dans le second cas corps et "tête" ne le seront pas. Ceci exige la redéfinition de CREATE (écran #24) et trois valeurs pour HEADFLG. Les deux premiers états sont mis explicitement par COMPILE-HEADS et DROP-HEADS, mais le troisième est reconnu et traité par CREATE.

→

Quand un mot est compilé, ses champs nom et lien sont compilés dans le dictionnaire des symboles, le mot est déclaré immédiat et (CFA) est compilé en tant que son champ code, suivi par l'adresse de l'emplacement mémoire suivant dans le dictionnaire principal. Le reste de la définition courante (le corps) sera alors compilé dans ce dernier. Lorsque le mot est appelé, son CFA est pris à l'emplacement mémoire suivant l'adresse du champ code de (CFA).

La fonction de (CFA) (écran #23) est soit de compiler l'adresse d'exécution du code dans le dictionnaire (lorsque le mot est ensuite utilisé dans une définitionn), soit d'exécuter la définition, ce qui dépend de STATE, avant d'"oublier" les symboles. L'implantation décrite ici traite le cas du 6502 et doit tenir compte du fait qu'aucun CFA ne doit être placé en XXFF, ce qui rend les définitions de (CFA) et CREATE un peu mystérieuses!

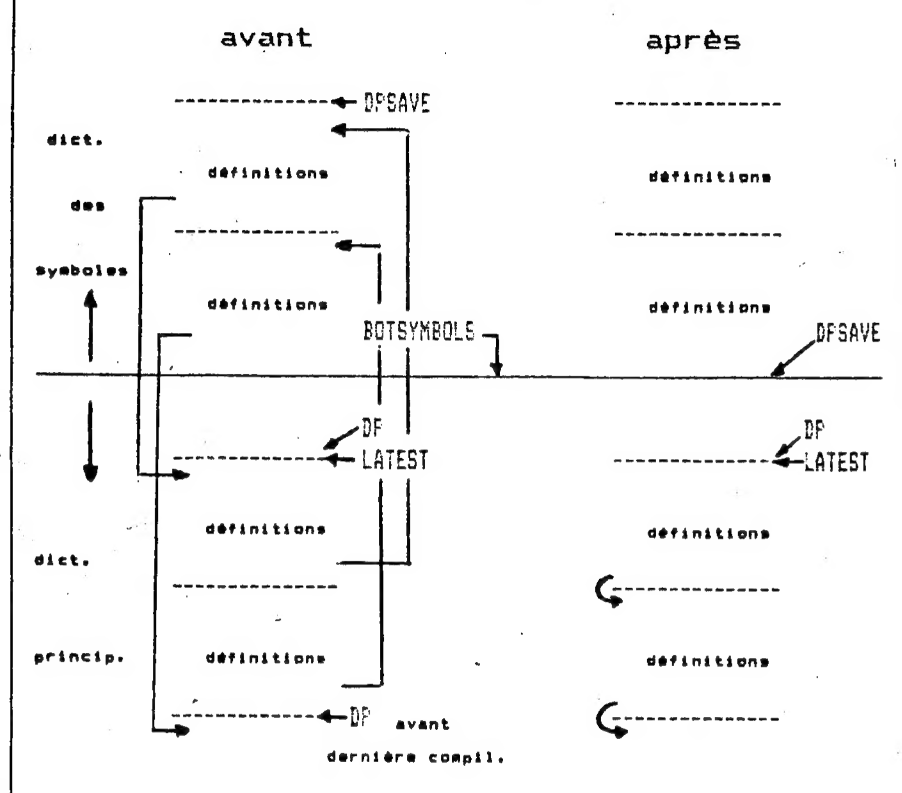


Figure 2

FORGET-SYMBOLS (écran # 22) parcourt chaque dictionnaire et "dé-lie" chacune des définitions placée dans le dictionnaire des symboles, en la libérant ainsi (fig. 2). C'est long, et on suppose qu'aucun symbole n'existe au-dessous de FENCE @ . Avant d'"oublier" quoi que ce soit dans le dictionnaire principal, vous <u>devez</u> appeler FORGET-SYMBOLS. Autrement des liens peuvent être rompus et l'interpréteur ne plus fonctionner.

L'étape suivante consistait à faire travailler aussi bien les mots de définition dans le mode DROP-HEADS, ce qui oblige à redéfinir (;CODE) (écran #25). Il utilise maintenant la sous-définition (;COD) et, selon l'état de HEADFLG, détermine l'emplacement du champ code à ré-écrire, et le ré-écrit.

Un problème peut se présenter dans le cas, rare, où une définition dont l'en-tête doit être "jetée" est supposée être immédiate. La "solution" est de déclarer ce cas comme étant illégal...Il y a toutefois une raison. Ce cas ne peut se produire que si la définition immédiate à placer dans le dictionnaire principal se trouve avec le mot [COMPILE] dans une définition. Autrement, il faudra de toute façon la compiler entièrement dans le dictionnaire des symboles. Un tel cas est si rare qu'il ne semble pas justifier l'effort de redéfinir IMMEDIATE et [COMPILE] . Pour utiliser un mot dont la tête a été compilée dans le dictionnaire des symboles en mode immédiat dans une définition, il faut faire .... [XXXX] ... !

Finalement, je remarquais que j'étais en train de créer <BUILDS ... DOES> et ;CODE avec des définitions à temps de compilation important, et ne faisant rien d'autre que d'utiliser de l'espace mémoire à l'exécution. Mais les parties "compilation" de ces définitions ne sont plus utilisées après compilation. Si donc on en supprime les en-têtes, tout ce qui précède DOES> peut être supprimé aussi bien. Il sera cependant nécessaire de redéfinir DOES> et ;CODE (écrans #26 et 27).

Au moment de la compilation, la situation d'une construction <BUILDS ... DOES> est la suivante: dans l'état DROP-HEADS, le nom a été placé dans le dictionnaire des symboles et en conséquence <BUILDS ... a été compilé dans le dictionnaire principal. Lorsque nous arrivons à DOES>, tout ce qui a été compilé dans ce dernier, y compris le champ code, doit être déplacé dans le dictionnaire des symboles.

Ceci est fait par MOVE-DEF? (écran #25), utilisé dans DOES> et ;CODE. Selon l'état de HEADFLG, MOVE-DEF? ou bien compile (;CODE), ou bien place la définition précédente dans le dictionnaire des symboles et compile ((;CODE)). ((;CODE)) doit avoir un pas d'indirection de plus que (;CODE) et a la même fonction que (CFA) dans les définitions ordinaires.

→

Une dernière remarque: les définitions de GOTO et LABEL, qui permettent des références avant multiples (par ex. plusieurs GOTOs) peuvent précéder aussi bien que suivre "leur" étiquette. Bien que je les aie implantées parce que c'était plus facile que la restructuration, on peut questionner leur valeur réelle, car elles occupent 318 octets!

GLOSSAIRE.

TOPDIC
constante empilant l'avant dernière adresse devant être utilisée comme dictionnaire principal.

BOTSYMBOLS
constante empilant la première adresse devant être utilisée comme dictionnaire des symboles.

TOPSYMBOLS
constante empilant l'avant dernière adresse devant être utilisée comme dictionnaire des symboles.

DPSAVE
variable contenant le pointeur vers le dictionnaire actuellement inactif.

TOD (Top Of Dictionnary)
variable contenant l'avant dernier emplacement mémoire courant à utiliser pour les définitions compilées.

MAIN-DIC
fait pointer DP sur le prochain emplacement mémoire libre du dictionnaire principal; les définitions qui vont suivre seront donc compilées en permanence dans le dictionnaire principal.
Si DP pointait déjà vers ce dictionnaire, il ne se passe rien.
Son contraire: SYMBOL-DIC.

SYMBOL-DIC
fait pointer DP sur le prochain emplacement mémoire libre du dictionnaire des symboles, c.à d. que les définitions qui suivent seront compilées dans ce dernier et pourront être effacées avec "FORGET-SYMBOLS" sans affecter le dictionnaire principal.
Si DP pointait déjà vers le dictionnaire des symboles, il ne se passe rien.
Son contraire: MAIN-DIC.

FORGET-SYMBOLS
est utilisé pour dé-lier les définitions compilées dans le dictionnaire des symboles à partir de celles du dictionnaire principal.
Replace le pointeur du dictionnaire des symboles sur "BOTSYMBOLS".
ATTENTION: si quelque chose a été compilé dans le dictionnaire des symboles, vous devez effectuer un "FORGET-SYMBOLS" avant d'effacer quoi que ce soit dans le dictionnaire principal.

HEADFLG
variable contenant l'état des en-têtes, c. à d.
HEADFLG = 0 --> COMPILE-HEADS a été effectué;
HEADFLG = 1 --> DROP-HEADS et MAIN-DIC ont été effectués;
HEADFLG = 2 --> DROP-HEADS et SYMBOL-DIC ont été effectués.

COMPILE-HEADS
compile les en-têtes (nom et champ de lien) des définitions qui suivent dans le dictionnaire principal.
Son contraire: DROP-HEADS.

DROP-HEADS
les en-têtes des définitions qui suivent vont être compilées dans le dictionnaire des symboles, le corps (champs code et paramètres) dans le dictionnaire principal. De plus, les parties compilation des définitions en terme de <BUILDS ... DOES> et ... ;CODE seront également compilées dans le dictionnaire des symboles,c.à d tout ce qui précède ... DOES> ou ... ;CODE dans la définition courante.Les parties situées dans le dictionnaire des symboles peuvent être effacées en effectuant "FORGET-SYMBOLS" qui rejette effectivement le code compilé des en-têtes. Tant que FORGET-SYMBOLS n'est pas effectué, ces mots peuvent être utilisés de façon normale, soit en compilation dans des mots de plus haut niveau, soit en exécution.
ATTENTION: ce mot ne doit pas être utilisé dans une définition "immédiate". Aucun contrôle d'erreur n'est réalisé!

→

```
SCR # 20
( SYMBOLDICTIONNARY                    KS 10-5-80 )
FORTH DEFINITIONS   HEX

3800 CONSTANT TOPSYMBOLS
3000 CONSTANT BOTSYMBOLS
3000 CONSTANT TOPDIC

3000 VARIABLE TOD
3000 VARIABLE DPSAVE

: SWITCH-DIC
   HERE  DPSAVE  DUP  @  DP  !  !   ;

: ?MAIN-DIC    ( --- F-1 )
   HERE  BOTSYMBOLS  U<  TOPSYMBOLS  HERE  U<  OR   ;
-->
```

```
SCR # 21
( SYMBOLDICTIONNARY                    KS 10-5-80 )

: MAIN-DIC
   TOPDIC   TOD  !  ?MAIN-DIC  0=  IF  SWITCH-DIC  THEN  ;

: SYMBOL-DIC
   TOPSYMBOLS   TOD  !  ?MAIN-DIC  IF  SWITCH-DIC  THEN  ;

: ?SYMBOL    ( N-1  ---  N-2,FLAG-1 )
   BOTSYMBOLS  OVER  1+  U<  OVER  TOPSYMBOLS  U<  AND   ;

: ?FENCE     ( N-1  ---  N-2,FLAG-1 )
   DUP  FENCE  @  U<   ;

-->
```

```
SCR # 22
( SYMBOLDICTIONNARY                    KS 10-5-80 )
: FORGET-SYMBOLS
   VOC-LINK  @
   BEGIN  DUP  @  >R  2 -  DUP  >R  @
          BEGIN  BEGIN  ?SYMBOL
                 WHILE  PFA  LFA  @
                 REPEAT  DUP  R>  !
                 BEGIN  PFA  LFA  DUP  @
                        ?SYMBOL SWAP  ?FENCE  ROT OR 0=
                 WHILE  SWAP   DROP
                 REPEAT SWAP   >R  ?FENCE
          UNTIL
          DROP  R>  DROP  R>  -DUP  0=
   UNTIL
   MAIN-DIC  BOTSYMBOLS  DPSAVE  !    ;
-->
```

```
SCR # 23
( SYMBOLDICTIONNARY                    KS 10-5-80 )

 0 VCARIABLE  HEADFLG

: COMPILE-HEADS    ?EXEC  0  HEADFLG  !   ;

: DROP-HEADS       ?EXEC  1  HEADFLG  !   ;

: (CFA)
   0  ,  HERE  0  ,  IMMEDIATE
   DOES>  @  STATE  @ -
   IF  ,  ELSE  EXECUTE   THEN    ;

-->
```

```
SCR # 24
( NEW CREATE                           KS 10-5-80 )
: CREATE    HEADFLG @
   IF  MAIN-DIC
       IF  2  ELSE  SYMBOL-DIC  1   THEN
       HEADFLG  !
   THEN
   TOD  @  HERE  0A0  +  U<  2  ?ERROR
   -FIND  IF  DROP  NFA  ID.  4  MESSAGE  CR   THEN
   HERE   DUP  C@  WIDTH  C@  MIN  1+  ALLOT
   DP  C@   0F0  =  ALLOT
   DUP  0A0  TOGGLE  HERE  1 -  080  TOGGLE
   LATEST ,  CURRENT  @  !  HEADFLG  @  1  =
   IF  0  HEADFLG  !  (CFA)  @  HEADFLG  !
       MAIN-DIC  DP  C@  0FF  =  ALLOT  HERE  SWAP  !
       TOD  @  HERE  2A0  +  U<  2  ?ERROR
THEN  HERE  2+  ,   ;    -->
```

```
SCR # 25
( MOVE-DEF?,                           KS 10-5-80 )
: (;COD)
   LATEST PFA HEADFLG @ 1 = IF @ ELSE CFA THEN !  ;

: (;CODE)    R>   (;COD)   ;
: ((;CODE))  R>  @  (;COD) ;

: MOVE-DEF?  ( rejette la partie <BUILDS )
   HEADFLG  @  1  =
   IF  SYMBOL-DIC  LATEST  PFA  DUP  CFA  DP  !
       @ DPSAVE @ >R DUP DPSAVE ! R> OVER - DUP
       HERE SWAP CMOVE R> ALLOT COMPILE ((;CODE))
       HERE  2  ALLOT  MAIN-DIC  HERE  SWAP  !
   ELSE  COMPILE  (;CODE)
   THEN    ;
-->
```

→

```
SCR # 26
( REDEFINITION OF <BUILDS  DOES>  KS 10-5-80 )

: <BUILDS
    CREATE   SMUDGE    ;

: DOES>
   MOVE-DEF?  020 C,  [ HERE 2 + ]  LITERAL , ; IMMEDIATE
   ASSEMBLER
   PLA,  TAY,  PLA,  N STA,  INY,  0=
   IF,  N  INC,  THEN,
   IP 1+  LDA,  PHA,  1 #  LDA,  PHA,
   IP STY,  N LDA,  IP 1+ STA,
   2 # LDA,  CLC,  W ADC,  PHA,
   0 # LDA,  W 1+ ADC,  PUSH JMP,

-->
```

```
SCR # 27
( REDEFINITION OF ;CODE          KS 10-5-80 )

;CODE
   ?CSP  MOVE-DEF?  [COMPILE]  [  SMUDGE
   !CSP  [COMPILE]  ASSEMBLER  ; IMMEDIATE

;S
```

```
SCR # 28
( GOTO                          KS 10-5-80 )
 FORTH DEFINITIONS  HEX
DROP-HEADS
: (GOTO)
   DOES>  DUP  @  BEGIN  -DUP
                  WHILE  DUP  @  SWAP
                         HERE  OVER  -  SWAP  !
                  REPEAT HERE  OVER  !
                  CFA  [ ' 0 CFA @ ]  LITERAL  SWAP  !  ;

: MOVE-HEAD      ( ---  HERE IN MAIN-DIC )
   HERE  SWITCH-DIC  DUP  HERE
   OVER  C@  WIDTH  C@  MIN  1+  DUP  >R  CMOVE
   HERE DUP 080 TOGGLE R>  ALLOT  DP  C@  0F0  =  ALLOT
   HERE  1-  080 TOGGLE  LATEST  PFA  LFA  DUP  @ , !    ;
-->
```

```
SCR # 29
( GOTO                            KS 10-5-80 )
COMPILE-HEADS
: GOTO
   COMPILE  BRANCH  -FIND
   IF  DROP  CFA @  [  ' 0 CFA @ ]  LITERAL  =
       IF  @  HERE  -  ,
       ELSE [ ' (GOTO) 2+ @ ] LITERAL OVER CFA @ =
            IF BEGIN DUP @ WHILE @ REPEAT
               HERE  SWAP  !  0 ,
            ELSE  4 ERROR
       THEN  THEN
   ELSE MOVE-HEAD [ ' (GOTO) 2+ @ ] LITERAL , ,
        SWITCH-DIC  2 ,
   THEN     ;     IMMEDIATE
-->
```

```
SCR # 2A
( GOTO                          KS 10-5-80 )

: LABEL     -FIND
   IF  DROP  CFA  DUP  @   [ ' (GOTO) 2+ @ ]   LITERAL   =
       IF  EXECUTE   ELSE  4  ERROR  THEN
   ELSE  MOVE-HEAD  [ ' 0 CFA @ ]  LITERAL  ,  ,  SWITCH-DIC
   THEN   ;    IMMEDIATE

  FORGET-SYMBOLS
;S

 ( les écrans 28 à 2A occupent 318 octets )
```

(Traduction A. I. fév. 86)

XIII LES TABLEAUX DE VARIABLES LOCALES

A) Introduction

Nous n'avons utilisé, jusqu'à présent, que des variables appelées variables simples (elles ne sont repérées que par leur nom). MUMPS permet de mettre en oeuvre des variables avec des indices de type "vectoriel", ou des variables à dimensions multiples (appelées "tableaux"). En fait, les variables de type "vectoriel" sont des variables isolées les unes des autres par un et un seul indice associé à un nom. Exemples :

NOM(1) NOM(2) NOM(3) ou NOM("DURAND") NOM("DUPONT") etc...

Pour les variables de type "tableau", la difference essentielle réside dans le fait que l'on cite plusieurs niveaux d'indices. Les exemples suivants illustrent des variables "tableau". Ne confondez pas "contenu" et "contenant". Pour le momment, nous ne parlons que des "contenants" (variables) puisque MUMPS ne fait de différence pour les "contenus" qu'au moment de leur utilisation.

NOM("SITUATION","DURAND","JEAN",1) ou NOM("SITUATION","DURAND","JEAN",2)

Dans l'exemple ci-dessus, la citation de la variable NOM descrit plusieurs niveaux ou dimensions de manière implicite et hiérarchique. La variable NOM("SITUATION","DURAND") existe en tant que telle parce que NOM("SITUATION") existe ou est censée exister. Les variables données pour exemple sont des variables à 5 dimensions. Les niveaux, hiérarchiques, utilisés sont les suivants :

```
NOM
NOM("SITUATION")
NOM("SITUATION","DURAND")
NOM("SITUATION","DURAND","JEAN")
NOM("SITUATION","DURAND","JEAN",nb)
```

MUMPS, lorsqu'il rencontre la dernière forme de citation, prend en compte automatiquement, s'ils n'existent pas, les niveaux supérieurs.

Ayant acquit un certain nombre d'éléments du langage MUMPS, nous allons analyser un petit programme simulant une saisie d'employés dont nous connaissons le numéro de sécurite sociale. En entrant uniquement au clavier un <cr>, la saisie est considérée comme terminée. La variable receptrice, dans ce cas, est alimentée avec une chaîne vide.

```
SAISIEMP  ;saisie des employés Y.L.G. 1/NOV/84 MAJ: 1/NOV/84
          ;SS est la variable contenant le No de S.S.
          S (SS,NBS)=0
DEBUT     R #,!,"No de sécurite sociale : ",SS G:SS="" FIN
          R !,"Nom de l'employé : ",NOM
          S FIC(SS)=NOM,NBS=NBS+1 G DEBUT
FIN       W !,"vous avez saisi : ",NBS," employés" Q
```

Pour cet exemple, nous avons volontairement choisi une programmation linéaire, afin d'en faciliter la lecture.

B) Les tableaux et leurs indices

Revenons au concept de tableau vehiculé par MUMPS. Contrairement à d'autres langages, l'utilisation de tableaux ne nécessite, à priori, pas de réservation en memoire. La place réellement utilisée par une variable tableau ne sera égale qu'à la place vraiment nécessaire. Quant aux indices, MUMPS ajoutera seulememt ceux qui n'existent pas dans la table des symboles. Cette particularité permet d'optimiser la place utilisée. Nous verrons ensuite la technique qu'utilise MUMPS pour stocker les variables ("simples" ou "tableau"). D'autre part, en MUMPS, le terme "tableau" est galvaudé puisqu'il est possible de renseigner uniquement certains éléments de celui-ci.

→

C) Restrictions sur les variables "tableau"

Le Standard MUMPS ne spécifie pas le type et la longueur des indices utilisables, mais pour des raisons de portabilité, il indique que les indices ne devront pas être des nombres négatifs et que la longueur définissant le tableau (nom de la variable + ses indices) n'exedera pas 64 caractères. Le contenu associé à cette variable ne depassera pas 255 caractères.

D) Exemple d'utilisation des tableaux en MUMPS

Imaginons que nous ayons besoin de créer un "fichier client". Nous pourrions écrire la routine suivante :

```
SAICLI    ;saisie d'un fic. client Y.L.G. 2/11/84 3/11/84
          S CLI="FICHIER CLIENT"
          F CLINO=1:1 D NOM,ADRESSE,CPOST D:NOM'="" CREA Q:NOM=""
FIN       Q  ;fin de programme
NOM       R !,"entrez le nom du client : ",NOM Q
ADRESSE   R !,"adresse du client : ",ADR Q
CPOST     R !,"code postal : ",CP Q
CREA      S CLI(CLINO)=NOM,CLI(CLINO,1)=ADR,CLI(CLINO,2)=CP Q
```

Le fait d'avoir attribué à la variable CLI le libellé FICHIER CLIENT revient à autodocumenter le fichier. Par la suite, nous avons alimenté le fichier avec deux niveaux d'indices. Le premier indice represente un numéro unique de client : valeur fournie par la variable de boucle. La donnée associée à cette variable est le nom du client. Puis, nous avons ajouté un deuxième niveau d'indices qui specifient dans un cas l'adresse, dans l'autre cas le code postal. Comme nous l'avons dit précédemment, MUMPS, lorsque nous citons le deuxieme indice, ne réserve plus de place pour l'indice précédent. En fait, il ne fait qu'attacher le nouvel indice cité à l'indice précédent.

E) La commande KILL utilisée pour les tableaux

La commande KILL que nous avons vue précédemment peut aussi être utilisée pour les tableaux. Lorsqu'elle est utilisée sans argument, KILL supprime toutes les variables "mémoire" avec ou sans indice. Si nous utilisons KILL nom de variable, KILL supprimera cette variable ainsi que tous ses indices s'ils existent. Il sera également possible de supprimer sélectivement des parties de tableaux. Par conséquent, si nous reprenons la variable CLI et que nous voulions seulement supprimer les informations concernant le client numero 10, il suffit d'écrire : KILL CLI(10).

A contrario, on peut également supprimer toutes les variables SAUF la variable concernant le client numéro 10. Dans ce cas, on ecrit : KILL (CLI(10)).

Les variables "tableau" dans lesquelles on trouve un indice ne devront être considerées que comme des variables ayant un ou des descendants (comme dans un arbre genealogique). La structure qu'utilise MUMPS pour stocker les tableaux est de type arborescent et hiérarchique.

F) Fonctions de manipulation de tableaux

MUMPS met a la disposition de l'utilisateur deux fonctions pour manipuler les tableaux. Avant d'utiliser une variable, il faut évidemment savoir si elle existe, ceci afin d'eviter les erreurs. Dans certains cas il est également nécessaire de savoir si une variable a des descendants.

F-1) La fontion $DATA

Le statut d'une variable peut être de quatre types :

→

- La variable n'est pas définie et n'a pas de descendant.
- La variable est définie mais n'a pas de descendant.
- La variable n'est pas définie mais a des descendants.
- La variable est définie et a des descendants.

Lorsque nous parlons de variables definies, nous sous-entendons que la variable a un contenu. Le rôle de la fonction $DATA consiste à permettre à l'utilisateur de connaître le statut d'une variable. Sa syntaxe est la suivante : $DATA(NOM DE VARIABLE) ou $D(NOM DE VARIABLE). Les valeurs retournées par cette fonction sont données dans le tableau suivant :

Valeur retournée	Explication
0	Variable indéfinie, sans descendant
1	Variable définie, sans descendant
10	Variable indéfinie, avec descendant(s)
11	Variable définie, avec descendant(s)

Pour exemple, considérons les variables suivantes :

A(1)="un"   A(1,3,5)="un trois cinq"   B="be"

Expression a evaluer	Valeur produite
$D(B)	1
$D(A)	10
$D(A(1))	11
$D(C)	0

F-2) La fonction $NEXT

Il est souvent intérressant de pouvoir parcourir une arborescence en ce qui concerne les différents indices. Cette fonctionnalité est réalisée grâce à $NEXT.

La fonction $NEXT retourne le prochain descendant hiérarchique de l'indice cité. Cette fonction est relativement difficile a comprendre. Afin d'être plus clairs, nous allons analyser ce qui se passe dans l'exemple ci-apres.

Supposons que nous ayons attribué à la variable X les indices suivants :

X(1) ; X(1,2) ; X(1,5,6) et X(2)

La valeur -1 sera retournée par $NEXT lorsqu'elle ne pourra plus délivrer d'autre indice du meme niveau. La valeur -1 permet également d'initialiser le sommet d'un niveau que l'on veut parcourir.

Si nous appliquons $NEXT aux variables précédemment définies, nous obtiendrons les résultats suivants :

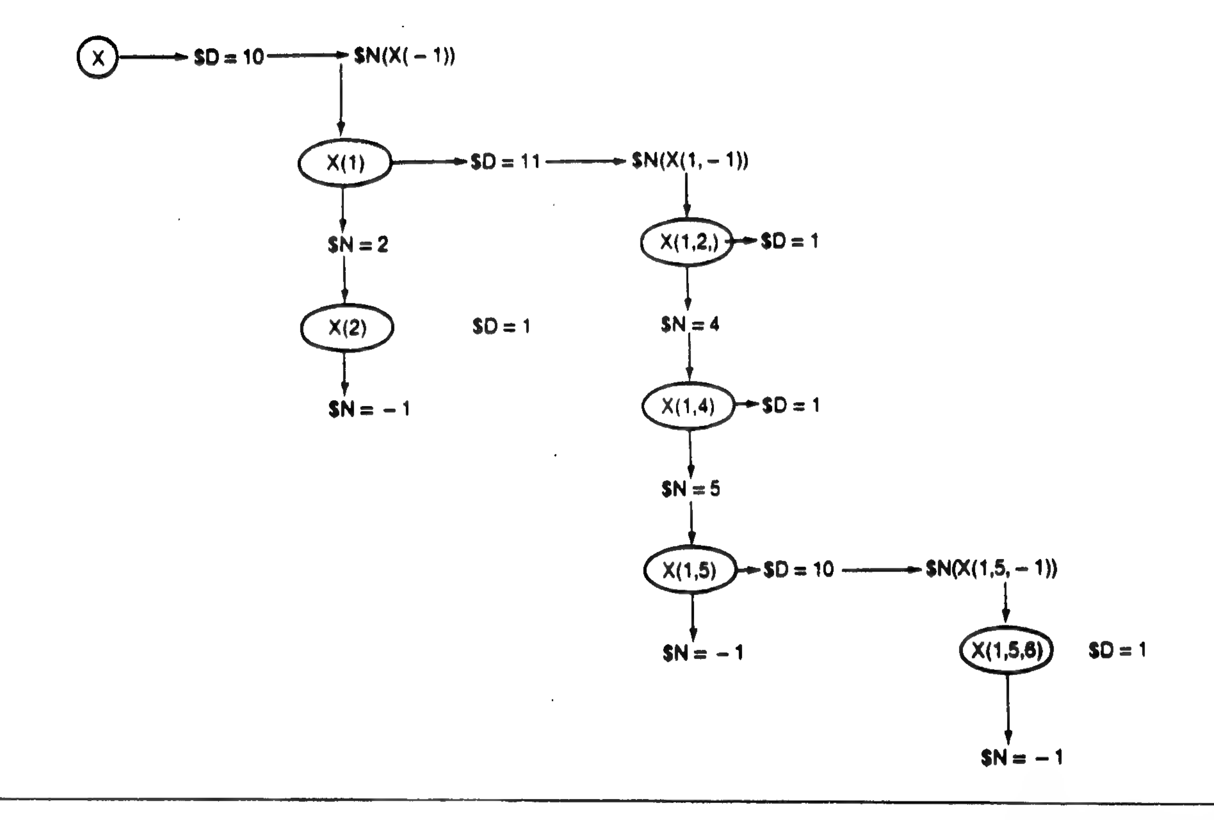
Variable	$NEXT(Variable)
X(-1)	1
X(1)	2
X(1,-1)	2
X(1,2)	4
X(1,5)	-1
X(1,5,-1)	6
X(1,5,6)	-1

RESUME

Ce chapitre nous a permis d'étudier une caractéristique importante de MUMPS et d'introduire deux nouvelles fonctions qui permettent de parcourir le tableaux emmagasinés par MUMPS. L'utilisation conjointe de $DATA et les $NEXT, illustrée ci-apres, demontre comme il est simple d'avoir l'image complète d'un tableau.

→

UTILISATION DE $NEXT ET DE $DATA



TRAITEMENTS DES ENSEMBLES

par Laurent BOUVOT

La vocation première de FORTH est de manipuler des nombres entiers, des chaînes de caractères et quelquefois des réels. Mais cela s'avère parfois insuffisant.

On peut, en effet, vouloir travailler avec d'autres types d'objets, tels les nombres complexes, les arbres et les listes, etc... Il faut alors étendre soi-même le vocabulaire FORTH.

C'est ce que j'ai été amené à faire pour manipuler les ensembles.

REPRESENTATION DES DONNEES

a) STRUCTURE D'UN ENSEMBLE.

Un ensemble est un groupement de mots (éléments) uniques et dont l'ordre n'a pas d'importance. On peut représenter les ensembles comme une chaîne de bits, où un 1 indique qu'un élément appartient à l'ensemble et un 0 indique qu'il ne lui appartient pas.

Cette représentation a plusieurs avantages:

- elle est économe en mémoire.
- la manipulation des ensembles sera relativement aisée.

Elle a cependant l'inconvénient d'être rigide. Un enseble vide occupe autant de place qu'un ensemble plein.

représentation d'un ensemble

No de l'octet	1 2 3
No du bit de l'octet	765432107654321076543...
indice de l'élément	876543218765432187654....

où chaque élément sera caractérisé par son indice (position de son bit représentatif dans l'ensemble). Exemple:

```
0110101100001000     est un ensemble
8765432187654321
```

qui comprend les éléments no 1,2,4,6,7,12, les éléments no 3, 5, 8, 9, 10, 11, 13, 14, 15, 16 n'appartenant pas à l'ensemble.

Pour manipuler des ensembles, il faudra donc créer des mots pouvant lire un bit ou écrire sur un bit.

b) REPRESENTATION

L'élément devra contenir la position du bit qui le représente, et devra en plus réserver un espace pour permettre à l'utilisateur de donner un contenu à cet élément.

L'élément sera constitué de deux zones, la première, appelée zone système qui est définie par:

- un octet contenant l'indice de l'élément, c'est à dire sa position dans l'ensemble.
- un octet indiquant la place disponible pour l'utilisateur (n mots de 16 bits).

Et la seconde zone, nommée zone utilisateur, constituée de n mots de 16 bits, où un élément peut représenter:

- une variable, une chaîne de caractère ou même un programme (Ndlr: on sent venir LOGO
- le nombre d'éléments, ainsi que le nombre de mots de 16 bits réservés à l'utilisateur. Cette quantité est limitée à 255 éléments.

c) CONVENTION D'ECRITURE

x: représente une suite d'octets sur la pile de longueur N.

N: représente le nombre d'octets d'une suite x sur la pile.

fl: représente le résultat d'un test; 0=échec, 1=succès.

s: représente le contenu d'un ensemble, c'est à dire une chaîne de bits.

e: représente l'indice d'un élément.

ADS: adresse du début du contenu de l'ensemble.

ADE: adresse du début du contenu de l'élément.

ORGANISATION DE S-FORTH

S-FORTH est composé d'un vocabulaire permettant la manipulation des ensembles, de deux tableaux et d'un certain nombre de constantes et de variables système.

a) LES TABLEAUX

Le tableau TES: pour chercher les ensembles contenant un élément, il faut connaître tous les ensembles existants, d'où le but de ce tableau, contenant toutes les adresses des ensembles. C'est, en quelque sorte, l'ensemble des ensembles.

Le tableau TEE: permet de faire la correspondance entre l'indice e d'un élément, et l'adresse ADE (début du contenu de l'élément).

L'indice est représenté par la position de ADE dans ce tableau. Exemple:

```
5 TEE @
```

donnera l'adresse du cinquième élément (e=5, ADE= 5 TEE @).

Ces deux tableaux sont unidimensionnels

b) LES CONSTANTES

Elles peuvent être modifiées par l'utilisateur avant initialisation de S-FORTH.

La constante NEM: contient le nombre maximum d'éléments définissables, valeur limitée à 255. Cette valeur définit la longueur de TEE ainsi que la longueur d'un ensemble.

La constante NSM: contient le nombre maximum d'éléments définissables. C'est en fonction de cette constante que la longueur du tableau TES sera calculée.

c) LES VARIABLES

Ce sont les variables du système S-FORTH.

La variable NED: contient le nombre d'éléments définis.

La variable NSD: contient le nombre d'ensembles définis.

Il sera nécessaire de connaître fréquemment la longueur d'un ensemble:

- en nombre d'octets.
- en nombre de mots de seize bits.



ce qui nous amène à définir deux constantes pratiques:

```
N8  = NEM / 8
N16 = NEM / 16
```

ce qui implique que NEM doit être un multiple de 16.

LE VOCABULAIRE S-FORTH

a) MANIPULATION DE LA PILE

Les objets à manipuler étant des ensembles, ils auront une longueur assez conséquente (N8 octets). Il faut donc prévoir des instructions pouvant agir sur plus de deux octets.

U? (--- adr)
donne l'adresse du sommet de la pile avant l'exécution du mot.

U! (adr ---)
donne au pointeur du sommet de la pile utilisateur la valeur adr.

R<D (n --- n)
copie le sommet de la pile de données sur la pile de retour.

PUL (n adr --- x)
empile les n octets contenus à partir de l'adresse adr.

REMOVE (n N ---)
enlève N octets de la pile à partir de l'adresse U? + N * n. Exemple:

```
xn+1 xn xn-1 ... x2 x1 x0 n N ----
--- xn+1 xn-1 ... x2 x1 x0
```

où xn a été enlevé de la pile.

NDROP (x N ---)
enlève N octets du sommet de la pile (après avoir enlevé N lui-même).

NDUP (x N --- x x)
duplique les N premiers octets de la pile.

NOVER (x1 x2 N --- x1 x2 x1)
même rôle que OVER mais sur des mots de N octets.

NPICK (x3 x2 x1 x0 3 N --- x3 x2 x1 x0 x3)

NSWAP (x2 x1 N --- x1 x2)

NROT (x3 x2 x1 N --- x2 x1 x3)

Ces mots sont d'utilité très générale. Ils permettent de manipuler des objets ayant un nombre d'octets supérieur à 4. A noter que l'on peut travailler sur un seul octet. Exemple:

1 NSWAP inverse le poids fort avec le poids faible d'un mot 16 bits.

SDROP (s ---)
supprime les N8 premiers octets de la pile, c'est à dire un ensemble.

SDUP (s --- s s)
duplique la chaîne de bits s (ensemble).

SOVER (s1 s2 --- s1 s2 s1)

SPICK (sn...s2 s1 s0 n --- sn...s2 s1 s0 sn)

SSWAP (s1 s2 --- s2 s1)

SROT (s3 s2 s1 --- s2 s1 s3)

b) ECRITURE ET LECTURE D'UN BIT

Les éléments étant représentés dans l'ensemble par un bit, il faut naturellement des instructions pouvant lire et écrire les bits d'un octet.

BIT? (n2 n1 --- b)
dépose sur la pile le contenu du nième bit de n2, n1 étant compris dans l'intervalle 0..7. Il est impossible de lire dans le poids fort de n2.

BIT! (n2 b n1 --- n)
charge le nième bit de n2 avec la valeur b, bétant égal à 0 ou 1 et n1 compris dans l'intervalle 0..7.

OCT>IND (n1 n2 --- numéro des bits à 1 de n2+1+8*(n+1))
dépose les numéros des bits qui sont à 1, additionnés de 8*(n1-1) et incrémentés de l'octet. Exemple:

```
3 % 11000111 --- 17 18 19 23 24
1 % 11100111 --- 1 2 3 6 7 8
```

c) CREATION

Les tableaux TES et TEE sont créés grâce à l'instruction ARRAY1. Attention, ARRAY1 ne fonctionne pas de la même façon suivant les machines utilisées. Exemple:

N ARRAY1 TABL	donne
soit n appartenant 0..N	longueur N+1
soit n appartenant 1..N	longueur N
soit n appartenant 0..N-1	longueur N

TES et TEE ont été définis de manière à réagir selon le second exemple.

ARRAY1 (N ---)
crée un tableau de longueur n mot(s) 16 bits dont l'indice est compris dans l'intervalle 1..N. Pour éviter toute erreur, sa définition est donnée dasn le vocabulaire S-FORTH.

CREATION D'UN ELEMENT:

EE (n ---)
crée un élément ayant n mots 16 bits de libre pour l'utilisateur. Exemple:

```
2 EE ELEMENT
```

crée un élément ayant pour nom ELEMENT, et réservant 4 octets pour l'utilisateur. Cet élément déposera son indice au sommet de la pile lors de chaque appel. Exemple:

```
ELEMENT .    affiche 1 OK
```

Pour connaître l'adresse de la zone libre de l'élément, il faudra se servir du tableau de correspondance:

```
ELEMENT TEE @ 2+
```

donne l'adresse du début de la zone utilisateur.

```
ELEMENT TEE @ 1+ C@
```

donne la longueur, en mots de seize bits, de la zone utilisateur.

CREATION D'UN ENSEMBLE

ES --- <mot>
permet la création d'un ensemble <mot>. Au départ, l'ensemble ayant un contenu quelconque, on prendra la précaution de le vider. Exemple:

```
ES ENSEMBLE
```

crée un ensemble nommé ENSEMBLE dont l'exécution déposera sur la pile l'adresse du début du contenu de l'ensemble.

→

d) OPERATIONS SUR LE CONTENU DES ENSEMBLES

ES. (ads ---) (qds=adresse ensemble)
affiche le contenu d'un ensemble.

VIDE! (ads ---)
vide un ensemble, c'est à dire met tous les bitsà zéro. Un ensemble créé par ES n'est pas vide au départ.

ENLEVE1 (e ads ---)
enlève l'élément d'indice e de l'ensemble d'adresse ads.

ENLEVE! ($FFFF e1 e2...en ads ---)
enlève les éléments dont les indices sont sur la pile, de l'ensemble d'adresse ads. *FFFF indique la fin de la liste des éléments à enlever.

APP1 (e ads ---) "appartenance"
rajoute l'élément d'indice e dans l'ensemble d'adresse ads, e appartenant à l'ensemble pointé par ads.

APP! ($FFFF e1 e2...en ads ---)
rajoute dans l'ensemble d'adresse ads les éléments sur la pile, ceci jusqu'au délimiteur FFFF (en hexa) soit 65535 en décimal.

CHARGE! (65535 e1 e2...en ads ---)
initialise un ensemble en le chargeant avec e1, e2...en. Cette instruction revient à écrire DUP VIDE! APP!. Le contenu de l'ensemble sera en,...,e2, e1.

)->
cette instruction est équivalente à CHARGE!. Seule la syntaxe change. Elle permet une meilleure lisibilité des programmes. Exemple:

65535 e1 e2 e3...en)-> ENSEMBLE

a pour effet de charger l'ensemble ENSEMBLE avec les éléments en...e3 e2 e1. 'ENSEMBLE ES.' donnera 'e1 e2 e3..en OK'.

<((--- 65535)
dépose la valeur 65535 (FFFF en hexa) sur la pile. Ainsi, '65535 e1 e2 e3 e4)-> A' est équivalent à '<(e1 e2 e3 e4)-> A'. Ceci accroit notablement la lisibilité du programme.

'ES@ (ads n --- x)
dépose sur la pile les n mots de seize bits contenus à partir de l'adresse ads.

'ES! (x ads n ---)
charge les n mots seize bits de la pile à partir de ads.

ES@ (ads --- s)
empile le contenu s de l'ensemble dont l'adresse ads est sur la pile.

ES! (s ads ---)
charge s dans l'ensemble d'adresse ads. Exemple:

```
ES S1 ES S2 ( crée les ensembles S1 et S2)
S1 ES@ S2 ES!
( S2 <- S1 => S1 = S2)
```

transfère le contenu de S1 dans S2.

e) OPERATIONS ENTRE LES ENSEMBLES

=! (ads1 ads2 ---)
remplit l'ensemble d'adresse ads1 avec le contenu de l'ensemble d'adresse ads2. Exemple:

```
S1 S2 =! est équivalent à
S2 ES@ S1 ES!
```

'OU (x1 x2 n --- x)
exécute un OU logique entre x1 et x2, soit deux chaînes d'octets de longueur n (x = x1 OR x2).

'ET (x1 x2 n --- x)
exécute un ET logique entre x1 et x2.

'OUEX (x1 x2 n --- x)
exécute un OU EXclusif entre x1 et x2.

'NON (x1 n --- x)
réalise un NON logique, c'est à dire une complémentation de x1 (x = NON(x)).

ET (s1 s2 --- s)
dépose sur la pile l'intersection des ensembles s1 et 2 (s = s1 inter s2).

OU (s1 s2 --- s)
dépose sur la pile le résultat de l'union des ensembles s1 et s2 (s = s1 union s2).

OUEX (s1 s2 --- s)
s= s1 XOR s2.

NON (s1 --- s)
s = NON(s)

Exemples:

```
S1 ES à  S2 ES à  OU  ES3 ES!
donne ES3 = S1 union S2

S1 ES à  NON S3 ES!
donne S3 = NON(S1)

S1 ES à  SDUP NON S2 ES à  OU ET S3 ES!
donne     S3 = ((.NON(S1) OU S2 ) ET S1 )
```
ou encore S3 = ((NON(S1) union S2) INTER S1)

f) OPERATIONS DE TEST

VIDE? (s --- f)
teste si un ensemble est vide. f=0 ensemble non vide; f=1 ensemble vide. Exemple:

S1 VIDE! S1 ES@ VIDE? . affiche 1 OK

APP? (e ads --- f)
teste si un élément d'indice e appartient à l'ensemble d'adresse ads. Exemple:

ELEMENT ENSEMBLE APP? .
donne 1 si ELEMENT appartient à ENSEMBLE
0 si ELEMENT n'appartient pas à ENSEMBLE

=? (s1 s2 --- f1)
teste si deux ensembles sont égaux: f1=0 si s1 est différent de s2; f1=1 si s1 est identique à s2. Exemple:

```
S1 ES@ SDUP =? .      affiche 1
S1 ES@ SDUP NON =? .  affiche 0
```

INCLU? (s1 s2 --- f1)
teste si s1 est inclut dans s2. f1=0 si s1 n'est pas inclut dans s2; f1=1 si s1 est inclut dans s2.

g) OPERATION DE RECHERCHE

LOCAL.E (e --- ADSn..ADS2 ADS1 n)
cette instruction recherche tous les ensembles contenant l'élément d'indice e. Elle dépose sur la pile l'adresse de ces ensembles et le nombre de ces ensembles.

LOCAL.S (ads --- adsn Aads-1..ads2 ads1 n)
cette instruction recherche tous les ensembles incluant l'ensemble d'adresse ads. Elle dépose sur la pile l'adresse de tous les ensembles dans lesquels l'ensemble d'adresse ads est inclut, ainsi que le nombre de ces ensembles.

NBE? (ads --- n)
indique le nombre d'éléménts contenus dans l'ensemble d'adresse ADS. Exemple:

S1 VIDE! S1 NBE? . affiche 0

→

EXEMPLE D'UTILISATION DE S-FORTH

Avec cette extension de FORTH, il est maintenant possible de manipuler des ensembles dont les éléments peuvent représenter n'importe quoi (variables, programme, etc...). Ces éléments sont liées entre eux par la relation APPARTIENT ou N'APPARTIENT PAS qu même ensemble. On peut donc réaliser des minis systèmes experts. Bien entendu, il ne s'agit pas de réaliser un système professionnel, mais d'expérimenter des raisonnements logiques.

Voici un exemple: on se propose de réaliser un programme devinant l'objet auquel le programmeur pense. Pour cela l'ordinateur considère au départ que tous les objets qu'il connait sont possibles. Puis il propose un ensemble. L'utilisateur devra alors dire si l'objet auquel il pense est contenu dans l'ensemble. A partir de là, la machine en déduira l'ensemble des éléments possibles et reproposera un ensemble. L'algorithme de recherche sera donc:

- soit SOL l'ensemble des éléments possibles
- soit EE? l'élément à deviner

```
 <( tous les éléments )-> SOL

1) propose un ensemble S1:  si EE? appartient à
S1 alors SOL <- SOL inter S1
   sinon SOL <- (SOL XOR S1) inter SOL
```

Si SOL ne contient qu'un élément ou s'il n'y a plus d'ensemble à proposer, alors FIN sinon boucle en 1). Bien entendu, le système ne proposera pas n'importe quel ensemble. Il testera d'abord si S1 inter SOL est différent de VIDE.

PROGRAMME

```
0 NED !   0 NSD !  ( initialise S-FORTH)
0 EE AVION      0 EE SCULPTURE 0 EE OURS
0 EE PLUMEAUX   0 EE CANARI    0 EE CHIEN
0 EE OREILLER   0 EE CORBEAU   0 EE LEZARD
0 EE AUTRUCHE

ES ANIMAUX      ES PLUMES      ES OBJETS
ES FAMILIER     ES SAUVAGES    ES UTILE
ES MAMMIFERE    ES OISEAUX     ES VOLE
ES SOL2         ES SOL

DECIMAL

<( OURS CHIEN CANARI CORBEAU AUTRUCHE
    LEZARD )-> ANIMAUX
<( AVION PLUMEAUX OREILLER SCULPTURE
    )-> OBJET
<( PLUMEAUX AVION OREILLER )-> UTILE
<( PLUMEAUX OREILLER CANARI CORBEAU
    AUTRUCHE )-> PLUMES
<( CHIEN CANARI )-> FAMILIER

FAMILIER ES@  NON ANIMAUX ES@
ET SAUVAGE ES!

<( AVION CANARI CORBEAU )-> VOLE

ANIMAUX ES@  PLUMES ES@  ET OISEAUX ES!
ANIMAUX ES@  OISEAUX ES@  NON
ET MAMMIFERE ES!

1 VARIABLE POINTEUR   0 VARIABLE S1

 : INITIAL
SOL2 VIDE!  SOL2 ES@  NON SOL ES!
1 POINTEUR !  0 S1 !  ;
( SOL2 est vide   SOL est plein=NON SOL2)

: DEMANDE ( s1 ---)
CR ." L'ELEMENT APPARTIENT-IL A CETTE LISTE"
CR SDUP  SOL2 ES!  SOL2 ES.
KEY  79 =
IF    SOL ES@  ET SOL ES!
ELSE  SOL ES@  SSWAP SOVER OUEX
      ET SOL ES!
ENDIF  ( OU THEN EN 79-STANDARD) ;

 : INTER?  ( s1 --- f1)
SOL ES@  ET VIDE?  0= ;

 : SOL?
S1 @  ES@  SDUP INTER?
IF      DEMANDE
ELSE    NON SDUP INTER?
  IF      DEMANDE
  ELSE    SDROP
  ENDIF
ENDIF ;
```

Thomson MO 5

```
 : DEVINE
 INITIAL
 BEGIN
   POINTEUR @  TES @  S1 !
   SOL? 1 POINTEUR +!  ( POINTEUR <- POINTEUR+1)
   POINTEUR @  NED @  >
   SOL NBE? 2 < OR
   ?TERMINAL  OR
 UNTIL
  ." SOLUTION:" SOL ES. CR ;
```

Une fois chargé, tapez DEVINE

LISTING DE L'EXTENSION S-FORTH

```
FORTH DEFINITIONS
VOCABULARY S-FORTH
( Le programme a été écrit sur un THOMSON MO5
avec la cassette FORTH de LORICIEL. Il est
donc possible que la création de mots en as-
sembleur ne soit pas standard. Dans ce cas, il
faut remplacer CREATE NOM par : NOM ;CODE )

HEX
CREATE U? 30C4 , 3610 , 0EB6 , SMUDGE

( EN ASSEMBLEUR:
 U?  LEAX ,U
     PSHU X
     JMP $B6    RETOUR A FORTH           )

CREATE U! 3710 , 3384 , 0EB6 , SMUDGE

( EN ASSEMBLEUR:
 U!  PULU  X    U<-U+2
     LEAU  ,X   U<-X
     JMP $B6                             )

CREATE NDROP ECC1 , 33CB , 0EB6 , SMUDGE

( EN ASSEMBLEUR:
 NDROP
     LDD  ,U++  D<-(U) ET U<-U+2
     LEAU D,U   U<-U+D
     JMP $B6                             )

CREATE R<D ECC4 , EDE3 , 0EB6 , SMUDGE

( EN ASSEMBLEUR:
 R<D LDD ,U       D<-(U)
     STD ,--S   POSE D SUR LA PILE SYSTEME
     JMP  $B6                            )
DECIMAL
 : PUL  ( n adr ---)
 >R   U?  2+ SWAP R<D  - R<D
  U! R> R> R> ROT ROT   CMOVE ;

 : REMOVE  ( n N ---)
 SWAP OVER *
 SWAP U? 4 +  DUP ROT +
 R<D ROT CMOVE-

  R> U! ;

  : NDUP  ( x N --- x x )
 U? 2+ PUL ;

  : NOVER  ( x1 x2 N --- x1 x2 x1)
 U? 2+ OVER + PUL ;

  : NPICK  ( xn..x0 n N --- xn..x0 xn)
 SWAP OVER * U?  4 + + PUL ;

  : NSWAP  ( x1 x2 N --- x2 x1)
 R<D NOVER
 2 R> REMOVE ;
```

→

```
 : NROT  ( x3 x2 x1 N --- x2 x1 x3)
>R 2 R PICK
3 R>  REMOVE ;

HEX
CREATE BIT? 3702 , 3702 , 3704 , 3704 ,
            4D C, 2704 , 544A , 20F9 ,
            C401 , 0EB4 , SMUDGE

( EN ASSEMBLEUR:
      PULU A
      PULU A
      PULU B
      PULU B
      TST A
DBTO BEQ FINO   SAUT EN FINO SI A=0
     LSRB       DECALE B VERS LA GAUCHE
     DECA       A<-A-1
     BRA DBTO
FINO ANDB #%00000001 ET LOGIQUE ENTRE B ET 1
     JMP *B4                               )

CREATE BIT! 3702 , 3702 , 3402 , 4D27 ,
            0566 , 434A , 20F8 , 86FE ,
            A443 , AA41 , 3344 , 3504 ,
            5D27 , 0449 ,  5A C, 20F9 ,
            1E89 , 0EB4 , SMUDGE

( EN ASSEMBEUR:
      PULU A
      PULU A
      PSHS A
      TSTA
DBT1 BEQ FIN1  SAUT EN FIN1 SI A=0
     ROR 3,U
     DECA
     BRA DBT1
FIN1 LDA #%11111110 A<-254
     ANDA 3,U
     ORA 1,U
     LEAU 4,U  U<-U+4
     PULS B
     TSTB
DBT2 BEQ FIN2
     ROLA
     DECB
     BRA DBT2
FIN2 EXG A,B
     JMP $B4                               )

DECIMAL
 : ARRAY1
<BUILDS 2* ALLOT DOES>
 SWAP 1- 2 * + ;

S-FORTH DEFINITIONS DECIMAL
80 CONSTANT NEM      80 CONSTANT NSM
 0 VARIABLE NED       0 VARIABLE NSD

NEM 8 /  CONSTANT N8
NEM 16 / CONSTANT N16

NEM ARRAY1 TEE      NSM ARRAY1 TES

 : EE ( Création d'un élément)
<BUILDS  1 NED +!  NED @  DUP C,
 OVER C, LATEST PFA 2+
SWAP TEE ! 2 * ALLOT
DOES>  C@  ;

 : ES ( crée un ensemble)
<BUILDS 1 NSD +! NEM 8 / ALLOT
LATEST PFA 2+ NSD @  TES !
DOES> ;

 : OCT>IND
SWAP 1- SWAP  8 0
DO
  DUP I BIT?
  IF
    OVER 8 * I + 1+
    ROT ROT
  ENDIF
LOOP
DROP DROP ;

 : << 65535 ; ( OU -1)

 : ES. ( ads ---)
65535 NEM 8 / 0
DO
  I 1+ 2 PICK
  I + C@   OCT>IND
  BEGIN     DUP 65535 = 0=
  WHILE     DUP NED@  >
    IF       DROP
    ELSE     TEE @  2 - NFA
    ENDIF
  REPEAT
LOOP DROP DROP ;

 : APP1  ( e ads ---)
SWAP 1 - DUP 8 / ROT
+ DUP C@  ROT 8 MOD
1 SWAP BIT! SWAP C! ;

 : VIDE!  ( ads ---)
NE? 8 / ERASE  ;

 : ENLEVE1  ( e ads ---)
SWAP 1 - DUP 8 / ROT
+ DUP C@  ROT 8 MOD
@ SWAP BIT!  SWAP C! ;

 : ENLEVE!  ( $FFFF e1..en ads ---)
BEGIN  OVER 65535 = 0=
WHILE  SWAP OVER ENLEVE1
REPEAT
DROP DROP ;

 : APP!  ( $FFFF e1..en ads ---)
BEGIN  OVER 65535 = 0=
WHILE  SWAP OVER APP1
REPEAT
DROP DROP ;

 : CHARGE!  ( $FFFF e1..en ads ---)
DUP VIDE! APP! ;

 : )->  -FIND
IF     DROP 2+ CHARGE!
ELSE   ." Ensemble inexistant" CR WARM ENDIF ;

 : =!  ( ads1 ads2 ---)
SWAP NEM 8 / CMOVE ;

HEX
CREATE 'ES@  3430 , ECC1 , AEC1 , 5D27 ,
             0810 , AE81 , 3620 ,  5A C,
             20F6 , 3530 , 0EB6 , SMUDGE

( EN ASSEMBLEUR:
'ES@       PSHS X,Y   EMPILE X ET Y SUR PILE SYS
           LDD ,U++   D<-(U);U<-U+2
           LDX ,UMM   X<-(U);U<-U+2
           TSTB       TEST DU CONTENU DE B
      DEB  BEQ FIN
           LDY ,X++   Y<-(X);X<-X+2
           PSHU Y     EMPILE Y SUR PILE UTIL
           DECB       B<-B-1
           BRA DEB
      FIN  PULS X,Y
           JMP $B6    RETOUR A FORTH         )

 : ES@   N16  'ES@   ;

CREATE 'ES!  3430 , ECC1 , AEC1 , 3A3A ,
             5D27 , 0837 , 2010 , AF83 ,
              5A C, 20F6 , 3530 , 0EB6 ,
               SMUDGE

( EN ASSEMBLEUR:
'ES!       PSHS X,Y   EMPILE X ET Y SUR PILE SYS
           LDD ,U++
           LDX ,U++
           ABX
           ABX        SOIT X<-X+2*B
           TSTB       TEST CONTENU DE B
      DEB  BEQ FIN
           PULU Y
           STY ,--X   X<-X-2;(X)<-Y
           DECB       B<-B-1
           BRA DEB
      FIN  PULS Y,X
           JMP $B6    RETOUR A FORTH         )
```

→

```
: ES!  N16  'ES!  ;

CREATE 'OU  3410 , ECC1 , 30C5 ,  5D C,
            2709 , A6C0 , AA84 , A780 ,
             5A C, 20F5 , 3510 , 0EB6 ,
                SMUDGE
```

(Pour le mot 'ET même programme; il suffit de remplacer AA84, càd ORA ,X, par A484, càd ANDA ,X)

(Pour le mot 'OUEX même programme; il suffit de remplacer AA84, càd ORA ,X, par A884, càd EOR ,X.)

```
( EN ASSEMBLEUR:
 'OU       PSHS X      EMPILE X SUR PILE SYS
           LDD ,U++
           LEAX B,U    X<-X+B
           TSTB        TEST DE B
     DEB   BEQ FIN
           LDA ,U+
           ORA ,X
           STA ,X+
           DECB
           BRA DEB
     FIN   PULS X
           JMP $B6     RETOUR A FORTH        )
```

(Pour 'ET remplacer ORA ,X par ANDA ,X)

(our 'OUEX remplacer ORA ,X par EOR A,X)

```
CREATE 'NON  3410 , ECC1 , 3040 , 5D27 ,
             0563 , 805A , 20F9 , 3510 ,
             0EB6 , SMUDGE

(  ASSEMBLEUR:
 'NON      PSHS X
           LDD ,U++
           LEAX 0,U  X<-U;
           TSTB
     DEB   BEQ FIN
           COM ,X+
           DECB
           BRA DEB
     FIN   PULS X
           JMP $B6     RETOUR A FORTH        )

DECIMAL
 : OU      N8 'OU ;        ( s1 s2 --- s)
 : ET      N8 'ET ;        ( s1 s2 --- s)
 : OUEX    N8 'OUEX ;      ( s1 s2 --- s)
 : NON     N8 'NON ;       ( s --- s)
 : SSWAP   N8 NSWAP ;      ( s1 s2 --- s2 s1)
 : SOVER   N8 NOVER ;      ( s1 s2 --- s1 s2 s1)
 : SDUP    N8 NDUP  ;      ( s1 --- s1 s1)
 : SPICK N8 NPICK ; ( sn..s0 n --- sn..s0 sn)

 : SROT    N8 NROT ;     ( s3 s2 s1 --- s2 s1 s3)
 : SDROP   N8 NDROP ;    ( s ---)

 : VIDE?  ( s --- fl)
1 N16 0 DO
  SWAP 0=  AND
LOOP ;

 : APP?  ( e ads --- fl)
OVER 1- 8 /
+ C@   SWAP 1-  8 MOD
BIT?  ;

 : =?  ( s1 s2 --- fl)
OUEX VIDE?  ;

 : INCLU?  ( s1 s2 --- fl)
SOVER ET =?  ;

 : LOCAL.E  ( e --- adsn..ads2 ads1 n)
0 NSD @  1+  1 DO
   OVER I TES @
   APP?
   IF     1+ I TES @   ROT ROT
   ENDIF
LOOP
SWAP DROP ;

 : LOCAL.S  ( ads --- adsn..ads1 n)
0 NSD @  1+  1 DO
  OVER DUP I TES @  =  0=
  SWAP ES@  I TES @  ES@
  INCLU?  AND
  IF     1+ TES @  ROT ROT
  ENDIF
LOOP
SWAP DROP ;

 : NBE?  ( ads --- n)
0 N8 0 DO
  8 0 DO
    OVER I + C@
    J BIT? I 8 *
    J + NED @ < AND
    IF    1+  ENDIF
  LOOP
LOOP
SWAP DROP ;
```

JEDI est une publication mensuelle éditée par l'ASSOCIATION JEDI
Association loi 1901. Cet exemplaire a été tiré à 500 exemplaires.
Président: Michel ROUSSEAU
Secrétaire général: Marc PETREMANN
Trésorier: Françoise BOLOTIN
Dépot légal: Préfecture de Paris
N° de Commission Paritaire: en cours

Achever d'imprimer sur les presses de:
S.A. ASHBAY COMMUNICATION, 162 rue du Fbg St HONORE 75008 PARIS

EDITEUR PLEIN ECRAN et COPIE GRAPHIQUE

par Roland JEANNIN

· Les deux utilitaires qui suivent constituent des atouts indispensables aux heureux utilisateurs du langage FORTH édité par JEDI pour le CPC d'AMSTRAD. Il s'agit d'un éditeur plein écran et d'un utilitaire de recopie d'écran graphique (hardcopy).

L'EDITEUR PLEIN ECRAN

Un éditeur en ligne sur un langage de quatrième génération pourrait être considéré comme un anachronisme. Mais en fait, cela contribue à la portabilité et à la simplicité du langage. Ainsi pourrons-nous conserver cette fonction tout en améliorant sensiblement la vitesse et le confort d'édition. Notre éditeur appelle au préalable pluisieurs remarques:

- la commande est tout simplement 'LISTP' au lieu de 'LIST'.
- le pavé des touches fléchées permet de se promener sur l'écran.
- quatres nouvelles touches sont émulées:
  - 'CLR' permet d'effacer les caractères sous le curseur.
  - 'ESC' permet de sortir de l'écran et de recopier dans le buffer. Ainsi n'y aura-t-il plus qu'à faire 'FLUSH' pour copier l'écran sur disque.
  - 'COPY' permet de mémoriser le caractère sous le curseur. On peut ainsi mémoriser jusqu'à 64 caractères, donc toute une ligne.
  - 'SHIFT'+'COPY' permet de restituer à l'emplacement du curseur toutes les lettres mémorisées dans la fonction 'COPY' et ceci en une seule fois. Cette fonction permet de réaliser un véritable 'coupé/collé'.
- on est d'office en mode insertion, ce qui facilite grandement les corrections d'erreurs. Une petite marge de dépassement de ligne est tolérée pour les corrections à l'affichage seulement.
- une petite astuce à remarquer est la constitution d'une véritable pile pour mémoriser la copie.
- il faut reconnaitre que l'affichage est un peu lent. Mais vous savez que vous pouvez entrer 20 caractères à la fois avant qu'ils ne soient affichés. Moyennant cela, vous pourrez jouir d'un confort inégalé à peu de frais en édition.
- la commande a été testée sur le CPC 664. Elle est très facilement adaptable aux autres types de CPC.

LA RECOPIE GRAPHIQUE

La recopie d'écran graphique est appelée par le mot 'HC'.
Quelques remarques seront utiles:

- pour compiler ce mot, on utilise un utilitaire constituant un véritable petit assembleur avec les possibilités suivantes:
  - définition de labels (**).
  - sauts absolus (JA) vers une adresse ou un label.
  - sauts relatifs (JR) en avant ou en arrière, de d octets ou vers un label.
  - simplification des procédures d'initialisation et de conclusion.
- la fonction pourra être appelée sous n'importe quel mode d'affichage.
- la recopie se fait pixel par pixel. L'imprimante est commutée automatiquement en mode graphique. Elle doit être du type EPSON (exemple: Centronics GLP, DMP 2000 ...).

Pour tout renseignement ou adaptation,
téléphoner à:
Mr Roland JEANNIN
au (16) 74.27.02.09 HB

AMSTRAD/SCHNEIDER

```
SCR # 35
 ( plein ecran page 1/4 )
 0 VARIABLE VFL ( flag d'etat de ligne ) 14 ALLOT
 0 VARIABLE NE ( numero d'ecran )
 0 VARIABLE V0 ( verticale initiale )
 0 VARIABLE H0 ( horizontale init. )
 0 VARIABLE AR ( pile supplementaire ) 64 ALLOT
 AR VARIABLE PA ( pointeur pile A )
 0 VARIABLE CTA ( compteur de A )
 HEX EE B4A6 C! E0 B49F C! EF B4EF C! DECIMAL
 ( affectation des touches du clavier pour cpc 664 )
 : >A ( empile de 1 octet sur A ) PA @ C! 1 PA +! ;
 : A> ( depile de 1 octet sur A ) -1 PA +! PA @ C@ ;
 : VF ( recherche dans table des flag ) V0 @ VFL + ;
 : V1 ( mise de flag a 1 ) 1 VF C! ;
 : LOCP ( locate reel ) SWAP 3 + SWAP 5 + LOCATE ;
 ;S
SCR # 36
 ( plein ecran page 2/4 )
 : ESC ( sortie d'ecran avec reecriture ) 16 0 DO I
        VFL + C@ 1 = IF 0 I VFL + C! I DUP 8 /MOD SWAP 64 *
        SWAP NE @ 2 * + BLOCK + SWAP 0 LOCP 64 0 DO DUP
        ?CHAR 9 EMIT SWAP I + C! LOOP DROP THEN UPDATE LOOP ;
 : INSERT ( c-- ) ( insertion ) 73 H0 @ DO ?CHAR SWAP EMIT LOOP
        DROP V0 @ H0 DUP @ 1+ DUP ROT ! LOCP V1 ;
 : EFF ( n=1 ou 2 ) ( fonction: sert a effacer ) H0 @ SWAP - 32
        SWAP 73 V0 @ OVER LOCP DO ?CHAR SWAP EMIT 8 EMIT 8 EMIT
       -1 +LOOP DROP V1 ;
 : HH ( sert a la fonction copie ) ?CHAR DUP >A 1 CTA +! EMIT
      1 H0 +! ;
 : RR ( sert a la fonction copie ) CTA @ DUP IF 0 DO A> LOOP
      CTA @ 0 DO INSERT LOOP 0 CTA ! ELSE DROP THEN ;
 : DEL ( effacer ) 2 EFF 9 EMIT -1 H0 +! ;
 : CLR ( effacer ) 1 EFF 9 EMIT ;     ;S
SCR # 37
 ( plein ecran page 3/4 )
 : ?C ( interrogation clavier ) ( --)
   BEGIN KEY DUP CASE
     11 OF V0 @ 0= IF 7 EMIT ELSE -1 V0 +! 11 EMIT THEN ENDOF
     10 OF V0 @ 15 = IF 7 EMIT ELSE 1 V0 +! 10 EMIT THEN ENDOF
     8  OF H0 @ 0= IF V0 @ 0= IF 7 EMIT ELSE V0 @ 1 - DUP V0 !
          63 DUP H0 ! LOCP THEN ELSE -1 H0 +! 8 EMIT THEN ENDOF
     9  OF H0 @ 63 = IF V0 @ 15 = IF 7 EMIT ELSE V0 @ 1+ DUP
      V0 ! 0 DUP H0 ! LOCP THEN ELSE 1 H0 +! 9 EMIT THEN ENDOF
     27 OF ESC ENDOF
     127 OF DEL ENDOF
     224 OF HH ENDOF
     238 OF CLR ENDOF
     239 OF RR ENDOF
     DUP INSERT ENDCASE
  27 = UNTIL 19 1 LOCATE ;     ;S
ok

SCR # 38
 ( plein ecran page 4/4 )
 : LISTP ( n-- ) ( commande principale et seul mot accessible )
        2 MODE DUP NE ! LIST 0 DUP V0 ! 0 DUP H0 ! LOCP ?C ;
 ;S
```

```
SCR # 41
( Langage Machine : calculs auto adresses + utilitaires )
: VAR VARIABLE ;   : CT CONSTANT ; 0 VAR LM80 0 VAR IDL 30 ALLOT
0 VAR IR 200 ALLOT  IR VAR PI  0 VAR CTI  : DEC DECIMAL ;
: >I PI @ ! 2 PI +! ;  : I> -2 PI +! PI @ @ ;  : IL 2 * IDL + ;
: LMD -2 ALLOT LATEST PFA DUP 2 - ! SP@ LMS0 ! HEX ;
HEX  0 VAR SPI LMD ED C, 7B C, LMS0 , C3 C, 12C ,
: LMC SP@ 2 - LMS0 @ 2 - 2DUP < IF DO I @ DUP ABS FF > IF , ELSE
 C, THEN -2 +LOOP THEN SPI ;
: LMF DEC C3 12C LMC CTI @ 1+ 0 DO I> DUP FF > IF
 100 - IL @ I> 2 - ! ELSE IL @ I> 1 - DUP ROT SWAP - 1 - SWAP
 C! THEN LOOP 0 CTI ! IR PI ! ;
: ** >R LMC HERE R> IL ! ;
: JR DUP >R LMC R> HERE >I >I 1 CTI +! ;
: JA 100 + JR ;  DEC

;S

SCR # 39
( copie d'ecran graphique = HC ) ( page 1/2 )
( compiler d'abord l'utilitaire de calculs langage machine )
0 VAR HC LMD
 C5 CD E JA ( saut en E ) C3 D JA ( saut en D )
 E ** CD BBBA CD BBE7 32 A JA CD 5 JA 21 018F 22 B JA 11 0 0
       3E 7 32 C JA
 0 ** CD 6 JA
 1 ** E 0 3A C JA 47
 2 ** E5 D5 C5 CD BBF0 C1 D1 21 A JA BE E1 37 20 3 JR A7
 3 ** CB 11 2B 2B 10 2 JR CD 9 JA 79 CD 8 JA 13 E5 21 027F 37
       ED 52 E1 38 4 JR 2A B JA 18 1 JR
 4 ** 23 7C B5 C8 2B 11 0 0 22 B JA 3E 7 BD 20 0 JR 7C B4 20
       0 JR 3E 4 32 C JA 18 0 JR
 5 ** 3E 1B CD 8 JA 3E 41 CD 8 JA 3E 7 CD 8 JA C9
 6 ** E5 3E 42 CD BB1E E1 28 7 JR E1 C9
;S

SCR # 40
( copie d'ecran graphique = HC ) ( page 2/2 )
 7 ** 3E D CD 8 JA 3E A CD 8 JA 3E 1B CD 8 JA 3E 4C CD 8 JA
       3E 7F CD 8 JA 3E 2 CD 8 JA C9
 8 ** CD BD2E 38 8 JR CD BD2B C9
 9 ** 3A C JA FE 7 C8 AF CB 11 CB 11 CB 11 C9
 A ** 0
 B ** 0 0
 C ** 0
 D ** C1
LMF
;S
```

FORTH HORLOGE TEMPS REEL

par J. CHANDRU

Le programme ci-joint, réalisé par un de nos adhérents, Mr J. CHANDRU, est un des premiers à utiliser le programme d'assemblage écrit en FORTH pour TO7, TO7-70.

Monsieur,

Je vous adresse (sans attente) mon horloge interne, sans attente d'une hypothétique amélioration de sa définition, à vous de voir si elle est digne d'intérêt en l'état.

Sa définition est de 100 ms. Son fonctionnement est perturbé par le LEP, la disquette, la sortie son. C'est un utilitaire permettant de comparer des temps d'exécution, et, éventuellement, de commander des exécutions insolites.

Les notations d'adresses déclarées en constantes sont les appelations du moniteur TO7.

TOP provoque la mise dans le pointeur d'IT de la première adresse routine ROHEUR; la mise à zéro de la variable chrono (TIME?); la sauvegarde du STATUS avant modification; la mise en route de l'horloge par modification du STATUS.

TIME affiche le chrono. Celui-ci repasse à zéro toutes les deux heures environ.

STOP arrête le chrono en restaurant l'ancien STATUS.

REMARQUE: Après essai, nous n'avons pas remarqué de dérive dans la mesure des temps, même sur des intervalles de mesures très longs, du style 30 minutes à 1 heure.

```
SCR: 20
HEX
E830 CONSTANT KBIN     ( sortie d'IT )
6019 CONSTANT (STATUS)
6027 CONSTANT (TIMEPT) ( pointeur IT)

VARIABLE STATUS?       (  memo status)
VARIABLE TIME?         (  chrono      )
CODE ROHEUR TIME? 1+ inc
                ne  if
            TIME?    inc then
            KBIN  jmp END-CODE
: TOP ( ---)
        ' ROHEUR (TIMEPT) !
               0 TIME?  !
   (STATUS) C@ DUP STATUS? C!
               20 OR (STATUS) C!   ;
: STOP ( ---)
           STATUS? C@ (STATUS) C!   ;
: :00 # 6 BASE ! # DECIMAL 3A HOLD ;
: TIME ( ---)
  TIME? @ 0
  <# # 2C HOLD :00 :00 #S #>  TYPE
  ."  Hr:Min:Sec" ;
 DECIMAL
```

Vous pouvez vous-même faire quelques essais:

```
: BC 10000 0 DO LOOP ;
```

puis

```
TOP BC TIME
```

affiche

```
00:00:00,6 Hr:Min:Sec
```

Faites vos propres essais en tapant:

```
COLORON CLS TOP VLIST TIME
```

et

```
COLOROFF CLS TOP VLIST TIME
```

Qu'en déduisez-vous ? ...

Thomson TO7-70